Symbiotic Scheduling of Concurrent GPU Kernels for Performance and Energy Optimizations

Symbiotic Scheduling of Concurrent GPU Kernels for Performance and Energy Optimizations

Teng Li
ECE Department
The George Washington University
tengli@gwu.edu

Vikram K. Narayana
ECE Department
The George Washington University
vikramkn@ieee.org

Tarek El-Ghazawi
ECE Department
The George Washington University
tarek@gwu.edu

ABSTRACT

The incorporation of GPUs as co-processors has brought forth significant performance improvements for High-Performance Computing (HPC). Efficient utilization of the GPU resources is thus an important consideration for computer scientists. In order to obtain the required performance while limiting the energy consumption, researchers and vendors alike are seeking to apply traditional CPU approaches into the GPU computing domain. For instance, newer NVIDIA GPUs now support concurrent execution of independent kernels as well as Dynamic Voltage and Frequency Scaling (DVFS). Amidst these new developments, we are faced with new opportunities for efficiently scheduling GPU computational kernels under performance and energy constraints.

In this paper, we carry out performance and energy optimizations geared towards the execution phases of concurrent kernels in GPU-based computing. When multiple GPU kernels are enqueued for concurrent execution, the sequence in which they are initiated can significantly affect the total execution time and the energy consumption. We attribute this behavior to the relative synergy among kernels that are launched within close proximity of each other. Accordingly, we define metrics for computing the extent to which kernels are *symbiotic*, by modeling their complementary resource requirements and execution characteristics. We then propose a *symbiotic* scheduling algorithm to obtain the best possible kernel launch sequence for concurrent execution. Experimental results on the latest NVIDIA K20 GPU demonstrate the efficacy of our proposed algorithm-based approach, by showing near-optimal results within the solution space of both performance and energy consumption. As our further experimental study on DVFS finds that increasing the GPU frequency in general leads to improved performance and energy saving, the proposed approach reduces the necessity for over-clocking and can be readily adopted by programmers with minimal programming effort and risk.

Categories and Subject Descriptors

C.1.3 [**PROCESSOR ARCHITECTURES**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; F.1.2 [**COMPUTATION BY ABSTRACT DEVICES**]: Modes of Computation—*Parallelism and concurrency*; F.2.2 [**ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*

General Terms

Algorithms, Design, Performance, Measurement

Keywords

GPU, Concurrent Kernel Execution, Performance, Energy, Scheduling, High-Performance Computing

1. INTRODUCTION

The proliferation of GPUs as application accelerators has been witnessed over the past few years in the HPC field. With the rapid advancements in the graphics technology, an increasing number of processing cores are integrated into the massively parallel processing architecture. As a result, a wide range of modern parallel computers have incorporated GPU accelerators to achieve well-improved floating-point performance beyond the Tera FLOPS (FLoating-point Operations Per Second) barrier. Such computers also include some of the world's fastest supercomputers such as Titan (Cray XK7), Tianhe-1A and Tsubame 2.0 [1].

Modern GPUs are composed of up to thousands of Streaming Processors (SPs), or CUDA cores, which are organized into Streaming Multiprocessors (SMs). The massive parallel architecture requires the application (kernel) to exhibit enough parallelism to fully utilize the computation power. However, since the internal parallelism expressed by a kernel heavily depends on the programming style as well as the problem size, device can be underutilized. As a result, improving the GPU resource utilization has been one of the major focuses for both GPU vendors and research communities. Concurrent kernel execution is one of such features provided available for both Kepler and Fermi series of GPUs by NVIDIA [15]. It improves the device utilization by allowing thread blocks from multiple kernels to be concurrently scheduled and executed on the SMs, provided that kernels are launched from separate CUDA streams. However, for both Fermi and Kepler series of GPUs, when there are several kernels ready for execution, all thread blocks from the earliest issued kernel are first assigned to the SMs, followed by thread blocks from the next issued kernel [18]. Although the Kepler GPU architecture provides an improvement over Fermi by using multiple hardware work queues (HyperQ) that help to eliminate false dependencies [15], we have observed from our experiments that the allocation of thread blocks to SMs still follows the kernel launch order. In other words, current GPU work distributor [15] (in charge of assigning thread blocks to SMs) on both generations of devices assigns thread blocks by following the kernel launch order, which determines how thread blocks from different kernels are co-scheduled on the SMs. Thread blocks from different concurrent kernels exhibit varied extent of SM resource usage (the usage of share memory, registers, warps and blocks etc.) as well as the utilization of both compute instruction throughput and global memory bandwidth. Launching independent kernels through individual CUDA streams allows all kernels to be queued for the distributor, which sequentially processes through the kernel queue and assigns thread blocks to an SM till one of SM resource limits is met. As a result, naively co-scheduled thread blocks from different kernels conflicting on one or more SM resources cannot be co-executed simultaneously within the same *execution round*. Instead, they will be executed in separate *execution rounds* and thus potentially underutilize the GPU. Note that we here define the term *execution round* as each round of simultaneous thread block execution on the SMs. (the same as the term *wave* [13] used by NVIDIA).

With a single modern high-end GPU consuming up to 300 watts, reducing the GPU energy consumption has also been an ongoing challenge, especially considering that a GPU-based supercomputer such as Titan consists 18,688 GPUs [1]. Vendors such as NVIDIA applied the Dynamic Voltage and Frequency Scaling (DVFS) technology to most of their GPUs, primarily for the purpose of reducing the idle energy consumption by lowering both the GPU and memory frequencies. However, under the scenario of concurrent kernel execution, optimizing for reduced GPU energy consumption at kernel run-time has been barely addressed by previous studies. Moreover, few previous works have been focused on studying the impact of frequency scaling on both performance and energy consumption for concurrent kernels. Therefore, it is imperatively necessary to explore run-time energy optimization techniques for concurrent kernel execution under GPU-based heterogeneous computing.

In this paper, we present an algorithm-based approach to optimize the performance and energy consumption of concurrent kernels being executed on the GPU. For both performance and energy optimizations, we focus our methodology on rescheduling the kernel launch sequence into multiple *symbiotic execution rounds* to eliminate potential SM resource contention among kernels. The term *symbiosis* [20] is used to refer the effectiveness of co-executing varied thread blocks from multi-kernels to achieve overall performance improvement, due to the elimination of SM resource confliction. To theoretically study the relative synergy among kernels that are launched within close proximity of each other, we define necessary resource utilization metrics for computing the extent to which kernels are *symbiotic*. A multi-threading model is presented to capture the utilization of compute instruction throughput and global memory bandwidth from multiple kernels and further utilized to derive the benefit of kernel *symbiosis*. Based on the theoretical modeling analysis, we propose a concurrent kernel *symbiotic* algorithm that reschedules the kernel launch order based on the *symbiosis* among kernels. The algorithm is designed to maximize the SM resource usage of each *execution round* as well as to appropriately balance the throughputs between compute instructions and global memory accesses. In other words, it can potentially optimize the performance and energy consumption by improving the GPU utilization among kernels. We therefore conduct a series of experiments by utilizing a single node of our Cray XK7 supercomputer with the latest NVIDIA K20 GPU. With different application benchmarks, our experimental results demonstrate that the proposed *symbiotic* algorithm is able to derive a kernel order schedule with near-optimal performance and energy results within all possible kernel orderings (the solution space). Furthermore, we experimentally study the impact of DVFS on concurrent kernel workloads and find that the best performance and run-time energy consumption comes from the highest GPU frequency. Thus, the proposed scheduling approach can potentially avoid over-clocking the GPU by still achieving an improved performance and energy efficiency compared with a random kernel schedule. To the best of our knowledge, our approach is the first to provide a performance/energy-aware solution for concurrent GPU kernels through kernel reordering and *symbiosis*. Moreover, the proposed work also has the potentials to impact the design of future GPU architecture and work distributor.

The rest of this paper is organized as the following: Section 2 provides an overview of the related work. Section 3 introduces a brief background of concurrent kernel execution and DVFS. Section 4 provides the resource metric modeling analysis as well as the detailed description of the proposed *symbiotic* algorithm. Section 5 presents the experimental results and further discussions, followed by the conclusion in Section 6.

2. RELATED WORK

There are several streams of research endeavors targeting at improving GPU resource utilization for the purpose of performance and power/energy optimization, respectively. However, researchers have overlooked the fact that the launch order of concurrent GPU kernels significantly affects the GPU SM resource utilization among multiple kernel blocks. In other words, it is counter-intuitive that the order of issuing concurrent kernels can affect the performance and energy consumption. A recent study [11] reported that the effect of kernel launch order on the performance side (total execution time) is trivial. However, their conclusion was erroneous because it was based on identical kernels differing only in the number of thread blocks (grid size) within each experiment. This "non-commutative concurrency" has only very recently been mentioned in [18] for GPUs. While their work follows a different approach through source-to-source transformation of kernels into kernels into elastic versions, our *symbiotic* algorithm solution identifies the source of the problem and requires little programming effort. As our method is mainly focused on reducing GPU resource competition among GPU kernels, similar research such as [6] also identifies the necessity to co-execute kernels with opposing resource boundaries (compute/memory). With a different focus, they primarily studied the possible performance gain with the kernel-merge approach specifically for OpenCL [10] kernels, due to OpenCL's lack of concurrent kernel support.

On the CPU side, [19] and [20] brought in the term "*symbiosis*" from the biology field and presented several predictors to possibly reduce microprocessor resource contention among time-slices of multiple OS workloads. While their studies are primarily concerned with co-executing time-slices of OS workloads on Simultaneous Multi-Threading (SMT) microprocessors, their experiments simply analyzed several CPU performance counters as the basis of co-executing workload time-slices. Note that we here merely borrow the term *symbiosis* and look at a completely different problem on how to efficiently co-schedule concurrent GPU kernels.

Several previous works have also studied possible optimization techniques for GPU energy improvement. Similarly as [6], [21] proposed a kernel-fusion method to fuse independent kernels to reduce energy consumption by improving GPU utilization. However, fusion/merging kernels requires kernel source code modification and imposes non-negligible overheads. Thus programmability and portability can be the one of the main issues, especially considering that modern GPU provides features such as concurrent kernel execution. [4] studied the energy consumption of single CUDA kernels with varied thread topology. Even though not directly stated, their results demonstrated that energy consumption can be improved by increasing the number of warps on the SM (warp occupancy) through changing the thread topology (different grid and block sizes). Studies such as [5] and [9] investigated the possible energy behavior with DVFS. [5] studied the effect of DVFS on the energy consumption for the latest K20 GPU. While only single kernel execution scenarios were studied, their work found that the highest clock setting on the K20 brings the best energy efficiency. Similarly, [9] experimentally analyzed DVFS for both compute and memory-bound kernels on an older GTX 280 GPU and observed that scaling down the GPU core frequency can save energy for memory-bound kernels. Furthermore, both [8] and [14] proposed the detailed GPU power modeling by using the performance counters and provided us with some optimization thoughts on the aspect of energy saving and future work. However, it is worth pointing out that none of these previous works has provided performance/energy optimization solution under the context of concurrent GPU kernels. We therefore believe that the work we present in this paper is the first to provide a concurrent kernel reordering/*symbiosis* solution aiming at optimizations on both performance and energy consumption.

3. BACKGROUND

Modern GPUs such as the NVIDIA Kepler K20 are composed of 13 SMs, each of which further consists 192 single-precision CUDA cores (SPs) and 64 double-precision units. Each SM executes one or more thread blocks simultaneously (up to 16 for Kepler) depending on the SM resource usages of a given SM. These SM resources include the shared memory usage; register usage of each thread and the total number of warps (a group of 32 threads) within the block. Note that warps are the atomic scheduling units within the SM. While multiple warps schedulers (4 for Kepler) are presented in each SM, each warp scheduler is composed of two instruction dispatch units, each of which issues one warp instruction per clock cycle. Since each SM has a limited number of registers, a fixed size of shared memory and a maximum number of warps that can co-exist and be scheduled, only multiple thread blocks meeting these resource constraints can be scheduled within a single SM.

Table 1: Memory and GPU Frequencies Supported by K20

Global Memory Clocks (MHz)	**Possible GPU Clocks (MHz) Under the Global Memory Clock**
2600 (Default)	758, 705 (Default), 666, 640, 614
324	324

Concurrent kernel execution provided by NVIDIA's Kepler and Fermi devices allows thread blocks from multiple kernels to be simultaneously executed on an SM. With a fixed kernel launch order sequence specified by the programmer using CUDA streams, the GPU work distributor processes through the kernels sequentially and assign the thread blocks on the SMs under a round-robin fashion until one SM resource limit is met. The remaining thread blocks are left for execution in the following *execution rounds* until all blocks are scheduled on the SMs. For a given single kernel or multiple kernels consisting of identical thread blocks (the number of blocks from each kernel does not need to be the same, note that this is also the only scenario studied and concluded by [11]), since each block requests identical amount of SM resources, the launch order does not affect the SM resource contention behaviors within each *execution round*. However, in the scenario of multiple non-identical kernels, which is usually the case, kernel launch order determines the consecutive thread blocks and thus the SM resource contention behavior. Therefore, we focus our analysis on finding the optimal kernel order, which minimizes the SM resource contention within each *execution round*. Note that, if the total number of blocks from all kernels does not exceed the number of SMs, the kernel launch order does not affect the SM resource utilization. In other words, there is no resource contention among blocks since each block has a dedicated SM. However, this scenario heavily underutilizes the GPU and therefore will not be considered in our optimization analysis.

For Fermi or earlier GPU architectures, SMs and SPs operate at different clocks while the clock of SPs generally doubles the clock of SMs. For the latest Kepler GPU architecture, SMs and SPs operate at the same frequency with many more SPs integrated into a single SM. Therefore, the Kepler architecture further expands the parallelism within each SM and thus greatly increases the available SM resources. Current DVFS technique provided by NVIDIA allows programmers to adjust the application clock settings for both GPU and global memory. As shown in Table 1, under two different memory clock states, only a few corresponding GPU clocks are supported for the latest Tesla K20. By using NVML API [16], the clock settings can be preset for a given GPU context (program). However, it is not yet possible to fine-control the frequencies within a CUDA program.

4. SYMBIOTIC SCHEDULING AND OPTIMIZATION

In this section, we are primarily focused on analyzing the SM resource utilization behavior of concurrent kernels. Since the optimization objective is to improve potential SM resource utilization when thread blocks from multiple kernels are presented, we focus our analysis on the following discussed two aspects. On one hand, due to the native multi-threading and high memory throughput architecture of GPUs, the utilization of compute instruction throughput

Table 2: Parameters Defined For Modeling Analysis and Algorithm

Symbol	**Definition** *
N_{SM}	The # of SMs in the GPU
N_{reg_SM}	The # of registers in each SM
S_{shm_SM}	The size of shared memory in each SM
N_{warps_SM}	The maximum # of warps allowed in each SM
N_{tblk_SM}	The maximum # of thread blocks allowed in each SM
IPC_{sp}	# of single-precision instructions per clock cycle assuming there is no memory latency for each SM
IPC_{dp}	# of double-precision instructions per clock cycle assuming there is no memory latency for each SM
IPC_{mem}	# of memory instructions per clock cycle for each SM
B_{max_mem}	The maximum global memory bandwidth (byte/cycle)
U_{gpu_b}	The optimum GPU utilization balance
N_{tblk_i}	The # of thread blocks for kernel i
$N_{thd\_per\_tblk\_i}$	The # of threads per block for kernel i
N_{inst_i}	The # of instructions for kernel i
N_{reg_i}	The # of registers for kernel i
N_{shm_i}	The size of shared memory used by kernel i
N_{warp_i}	The # of warps for kernel i
$N_{warp\_SM\_i}$	The # of warps per SM for kernel i
$N_{tblk\_SM\_i}$	The average # of thread blocks per SM for kernel i
R_{sp_i}	The single-precision instruction ratio for kernel i
R_{dp_i}	The double-precision instruction ratio for kernel i
R_{mem_i}	The memory instruction ratio for kernel i
N_{mem_i}	The average memory access size (byte) for kernel i
B_{mem_i}	The memory bandwidth (byte/cycle) for kernel i
$T_{avg\_data\_i}$	The average data access latency (cycle) for kernel i
IPC_{max_i}	The # of compute instruction (per SM, including single and double precision) per cycle on average for kernel i assuming there is no memory latency
IPC_{max_r}	The # of compute instruction (per SM, including single and double precision) per cycle on average for *execution round* r assuming there is no memory latency
U_{ipc_i}	The compute instruction throughput (IPC) utilization for kernel i
U_{memb_i}	The memory bandwidth utilization for kernel i
U_{ipc_r}	The IPC utilization for *execution round* r
U_{memb_r}	The memory bandwidth utilization for *execution round* r
$U_{gpu\_b\_i}$	The GPU utilization balance for kernel i
$U_{gpu\_b\_r}$	The GPU utilization balance for *execution round* r
$U_{gpu\_b\_devlim}$	The Deviation limit on the GPU utilization balance

*The upper parameters are constant for a given GPU, whereas the remaining parameters are application dependent.

(from many threads) as well as the global memory bandwidth are inter-correlated and need to be balanced. In other words, unbalanced utilizations can result in either memory-bound or compute-bound behavior. Therefore, co-scheduling kernels with opposing compute/memory boundaries can improve and balance both compute and memory throughputs. On the other hand, optimizing kernel co-schedules to utilize all GPU device (SM) resources including shared memory, registers and co-existing warps can increase the total number of warps within each *execution round* (warp occupancy) as well as possibly reduce the number of *execution rounds*. Therefore, we here initially present a multi-threading model to capture the utilizations of both compute instruction throughput and memory bandwidth. The modeling analysis is further utilized in the *symbiotic* algorithm as one of the decision-making factors.

4.1 Modeling Analysis on Compute/Memory Utilization and Balance for Concurrent Kernels

As each SM of the GPU can keep track of many threads within an *execution round*, the very low context-switch overhead among threads allows the execution of compute instructions from many threads to hide the global memory access latency. Thus, under concurrent kernel execution, different thread blocks (from different kernels) with varied ratios between the number of compute and memory instructions are to be executed within each required SM *execution round*. By extending the multi-threading principles studied by [7], we here derive a GPU utilization model to predict the utilizations of both compute instruction throughput and memory bandwidth as well as the *GPU utilization balance*, which is the metric defined to measure the ratio between the utilizations of compute instruction throughput and memory bandwidth. As Table 2 defines necessary parameters for our modeling analysis, we initially analyze the scenario with a single kernel i to be executed and further extend the analysis to cover an *execution round* with multiple kernels. Note that we focus our analysis on a single SM due to the SPMD model adopted by GPUs. We also assume that there is no dependency among all threads and context-switch overhead among threads is negligible.

For a given kernel i composed of memory, single-precision and double-precision instructions, a thread need to stall when executing a memory instruction so that other threads can switch in. Assuming $R_{mem\_i}+R_{sp\_i}+R_{dp\_i}=1$, for every $1/R_{mem\_i}$ instructions on average, a thread needs to stall to wait for the global memory access. With the multi-threading architecture of each SM, stalled threads are filled with other threads executing compute instructions. Since the average maximum Instruction Per Cycle (IPC) for compute instructions can be derived from (1), for each SM, the number of extra threads required to switch in to fully hide the data access latency is $T_{avg\_data\_i} \cdot R_{mem\_i} \cdot IPC_{max\_i}$. Note that $T_{avg\_data\_i}$ defines the average number of clock cycles to access the global memory. As one of our purposes to conduct the multi-threading modeling is to demonstrate a simple way of deriving the compute instruction throughput utilization U_{ipc_i}, here for simplicity, we do not take the impact of cache on $T_{avg\_data\_i}$ into consideration.

$$\begin{aligned} IPC_{max\_i} &= \frac{R_{sp\_i} + R_{dp\_i} + R_{mem\_i}}{R_{sp\_i}/IPC_{sp} + R_{dp\_i}/IPC_{dp} + R_{mem\_i}/IPC_{mem}} \\ &= \frac{1}{R_{sp\_i}/IPC_{sp} + R_{dp\_i}/IPC_{dp} + R_{mem\_i}/IPC_{mem}} \end{aligned} \tag{1}$$

Considering the current thread issuing the memory instruction along with all other additional threads needed in order to fully hide memory latency, a total number of $(T_{avg\_data\_i} \cdot R_{mem\_i} \cdot IPC_{max\_i} + 1) \cdot N_{SM}$ threads will satisfy the memory latency hiding requirement and thus achieve IPC_{max_i} theoretically. Therefore, based on the number of threads requested by kernel i, which equals to $N_{tblk\_i} \cdot N_{thd\_per\_tblk\_i}$, compute (IPC) utilization of kernel i can be derived from (2). Note that here we assume that kernel i can be fit within a single *execution round*.

$$U_{ipc\_i} = MIN\left(\frac{N_{tblk\_i} \cdot N_{thd\_per\_tblk\_i}}{(T_{avg\_data\_i} \cdot R_{mem\_i} \cdot IPC_{max\_i} + 1) \cdot N_{SM}}, 1\right) \tag{2}$$

For the memory bandwidth of kernel i, $R_{mem\_i} \cdot N_{inst\_i}$ memory instructions need to request $R_{mem\_i} \cdot N_{inst\_i} \cdot N_{mem\_i}$ bytes of data within $\frac{N_{inst\_i}}{IPC_{max\_i} \cdot U_{ipc\_i} \cdot N_{SM}}$ cycles. Therefore B_{mem_i} can be derived with (3). Accordingly, the utilization of memory bandwidth for kernel i, U_{memb_i}, can be calculated using (4).

$$B_{mem\_i} = \frac{R_{mem\_i} \cdot N_{inst\_i} \cdot N_{mem\_i} \cdot IPC_{max\_i} \cdot U_{ipc\_i} \cdot N_{SM}}{N_{inst\_i}} = R_{mem\_i} \cdot N_{mem\_i} \cdot IPC_{max\_i} \cdot U_{ipc\_i} \cdot N_{SM} \quad (3)$$

$$U_{memb\_i} = \frac{B_{mem\_i}}{B_{max\_mem}} = \frac{R_{mem\_i} \cdot N_{mem\_i} \cdot IPC_{max\_i} \cdot U_{ipc\_i} \cdot N_{SM}}{B_{max\_mem}} \quad (4)$$

Furthermore, to demonstrate the balance between the utilizations of compute throughput and memory bandwidth, we use the metric $U_{gpu\_b\_i}$ and derive it with (5).

$$U_{gpu\_b\_i} = \frac{U_{ipc\_i}}{U_{memb\_i}} = \frac{B_{max\_mem}}{R_{mem\_i} \cdot N_{mem\_i} \cdot IPC_{max\_i} \cdot N_{SM}} \quad (5)$$

In our modeling analysis, we are mostly interested in $U_{gpu\_b\_i}$ and will further use it as one *symbiosis* metric in our algorithm. While $U_{gpu\_b\_i}$ is able to demonstrate the compute/memory utilization boundaries of each kernel, (kernel i is memory-bound when $U_{gpu\_b\_i} < 1$ and vice versa.) correlating and combining the metric of $U_{gpu\_b\_i}$ from multiple kernels (further defined as $U_{gpu\_b\_r}$) for each *execution round* helps us better understand the overall compute and memory utilization balance when kernels are concurrently executed. It also provides us insights on co-scheduling kernels with opposing resource requirements. We therefore extend the modeling analysis to multiple kernels and are focused on the compute/memory utilization analysis for a single *execution round* composed of multiple kernels. This is because thread blocks from multiple kernels are executed under the multi-threading model for each *execution round.* Therefore, assuming m kernels can be executed within an *execution round*, IPC_{max_r} can be derived with (6) by considering N_{inst_i} of each kernel within the *execution round.* By following (5) and combining the contribution of each kernel, the compute/memory utilization balance for a specific *execution round*, $U_{gpu\_b\_r}$, can be derived using (7). Note that U_{ipc_r} can be derived similarly as (2) for multiple kernels. Since we are primarily interested in studying $U_{gpu\_b\_r}$, which is not directly correlated to U_{ipc_r} similar as (5), the detailed deriving equation for U_{ipc_r} is not discussed here.

$$IPC_{max\_r} = \frac{\sum_{i=1}^{m} N_{inst\_i}}{\sum_{i=1}^{m} \left(R_{sp\_i}/IPC_{sp} + R_{dp\_i}/IPC_{dp} + R_{mem\_i}/IPC_{mem}\right) \cdot N_{inst\_i}} \quad (6)$$

$$U_{gpu\_b\_r} = \frac{B_{max\_mem} \cdot \sum_{i=1}^{m} N_{inst\_i}}{IPC_{max\_r} \cdot N_{SM} \cdot \sum_{i=1}^{m} R_{mem\_i} \cdot N_{inst\_i} \cdot N_{mem\_i}} \quad (7)$$

4.2 Optimization Strategies

Since our optimization goal is mainly focused on improving the GPU resource utilization, we concentrate on two utilization aspects while setting the optimization unit to be each *execution round.* This is because kernel blocks are co-executed on SMs through multiple consecutive *execution rounds.* The previously discussed multi-threading model captures the utilization balance of compute instruction throughput and memory bandwidth within each *execution round.* By considering the analyzed utilization balance, one of our optimization strategies is to optimize each *execution round* with the minimum difference in the utilization ratios of both compute and memory bandwidth. This is due to the fact that a kernel or an *execution round* profiled as either compute or memory-bound generally has the underutilized memory/compute resource, which can greatly impact the overall performance as well as energy consumption.

Based on the previous modeling analysis, in order to improve the utilization of either compute throughput or memory bandwidth, more corresponding instructions need to be provided by more threads within the *execution round.* Therefore, we focus the other optimization strategy on improving the number of warps (threads) that can be co-executed. Under the current GPU architecture, the number of threads blocks (from multiple kernels) that can be executed within each execution round is limited by the SM hardware resource limitations including N_{reg_SM}, S_{shm_SM}, N_{warps_SM} and N_{tblk_SM} as defined in Table 2. In other words, for each *execution round*, meeting either one of these resource limitations will block further thread blocks to be scheduled in. As we discussed earlier, for a sequence of concurrent kernels with varied profiles, different grid/block configurations and the amount of shared memory requested by each kernel determines the number of *execution rounds* as well as the number of different blocks within each *execution round.* Therefore, consecutive kernels competing for the same resource (for example, two consecutive kernels both requesting a large shared memory size) can be delayed with extra *execution rounds.* The resource underutilization can happen when any of the consecutive kernels barely requires other SM resources. Thus, we believe a fundamental solution to tackle this problem without complicated kernel source modification is to design an algorithm which can derive an SM resource-optimized kernel launch order with balanced utilization of compute throughput and memory bandwidth.

4.3 Concurrent Kernel Symbiotic Algorithm

The fundamental concept of scheduling kernel launch order allows each kernel to be co-executed with different other kernels within each *execution round.* Therefore, two or more kernels with opposing resource requirements can be mutually beneficial in terms of resource exploitation. While the term *Symbiosis* is defined as the mutually beneficial living together of two dissimilar organisms in close proximity [19], we here adapt this term to refer the improvements of the SM resource utilization by co-scheduling thread blocks from two or more kernels. As a result, possible reduction on both execution time and energy consumption can be achieved. Here we present and implement a resource-aware *symbiotic* algorithm that provides a suitable launch order for a sequence of concurrent kernels. One fundamental *symbiosis* consideration is to maximize the number of kernels that can be executed within a single *execution round.* In other words, this allows a possibly large number of threads for each SM to schedule and execute simultaneously in each *execution round.* In other words, our desired algorithm should maximally eliminate kernel blocks' confliction on the SM hardware resources and thus improve the warp occupancy of each *execution round.* The other optimization consideration focuses on balancing the utilization of compute throughput and memory bandwidth of a given *execution round.* In other words, we desire an algorithm that can schedule the launch order of kernels so that each *execution round* is *symbiotically* optimized with balanced compute/memory utilization and eliminated SM hardware resource confliction.

```
Input: the set of $N_{knl}$ kernels ($K$) with profiling results ($PR$):
    $N_{tblk\_i}$, $N_{reg\_i}$, $N_{shm\_i}$, $N_{warp\_i}$, $U_{gpu\_b\_i}$
Output: Kernel order from symbiotically optimized $Rd_1$ to $Rd_r$
    ...........................................................
    Denote $Rd_r$ to be the set storing symbiotically optimized kernel order within execution round $r$; r=0
    SymbiosisScoreMatrix[ ][ ]=SymbiosisScoreGen($K$, $K$, $PR$)
    while $K$ != null do
 5:     r++                ▷ Counting towards the next execution round
        Within $K$, find kernel: $K_a$,$K_b$ with the highest score in SymbiosisScoreMatrix[ ][ ]
        Push $K_a$,$K_b$ into $Rd_r$ (using the decreasing order of $N_{shm\_a}$, $N_{shm\_b}$) and remove $K_a$ and $K_b$ from $K$
        $K_{comb}$=ProfileCombine($K_a$,$K_b$)
        for All kernels $K_r$ (from $K$) whose utilized SM resource can fit within $Rd_r$ do
10:         SymbiosisScoreVec[ ]=SymbiosisScoreGen($K_{comb}$, $K_r$, $PR$)
            Push $K_c$ with the highest score in SymbiosisScoreVec[ ] into $Rd_r$ (Sort by $N_{shm\_c}$, $N_{shm\_comb}$ with decreasing order)
            $K_{comb}$=ProfileCombine($K_{comb}$,$K_c$) and remove $K_c$ from $K$
    ...........................................................
    function SymbiosisScoreGen($K_M$, $K_N$, $PR$)   ▷ $K_M$ & $K_N$ are two kernel sets
15:     for All kernels $K_i$ within $K_M$ do
            for All kernels $K_j$ within $K_N$ do
                if $K_i$ and $K_j$ cannot fit within an execution round then S[i][j] = 0
                else
                    S[i][j] += max{$\frac{N_{shm\_SM} - N_{shm\_i} - N_{shm\_j}}{N_{shm\_SM}}$, 0}
20:                 S[i][j] += max{$\frac{N_{reg\_SM} - N_{reg\_i} - N_{reg\_j}}{N_{reg\_SM}}$, 0}
                    S[i][j] += max{$\frac{N_{warp\_SM} - N_{warp\_i} - N_{warp\_j}}{N_{warp\_SM}}$, 0}
                    S[i][j] += max{$\frac{N_{tblk\_SM} - N_{tblk\_SM\_i} - N_{tblk\_SM\_j}}{N_{tblk\_SM}}$, 0}
                    if $U_{gpu\_b\_i} \leq U_{gpu\_b} \leq U_{gpu\_b\_j}$ or $U_{gpu\_b\_j} \leq U_{gpu\_b} \leq U_{gpu\_b\_i}$ then
                        if $U_{gpu\_b\_comb(i,j)} = U_{gpu\_b}$ then
25:                         S[i][j]+= 1        ▷ $U_{gpu\_b\_comb(i,j)}$ is the combined utilization balance
                        if $U_{gpu\_b\_comb(i,j)} > U_{gpu\_b}$ then
                            S[i][j]+= max{$(1 - \frac{U_{gpu\_b\_comb(i,j)}}{U_{gpu\_b} \cdot U_{gpu\_b\_devlim}}), 0$}        ▷ $U_{gpu\_b\_devlim}$ set the upper limit on $U_{gpu\_b\_comb(i,j)}$
                        if $U_{gpu\_b\_comb(i,j)} < U_{gpu\_b}$ then
                            S[i][j]+= max{$(1 - \frac{U_{gpu\_b}}{U_{gpu\_b\_comb(i,j)} \cdot U_{gpu\_b\_devlim}}), 0$}        ▷ $U_{gpu\_b\_devlim}$ set the lower limit on $U_{gpu\_b\_comb(i,j)}$
        return S[ ][ ]
30: end function
    ...........................................................
    function ProfileCombine($K_a$, $K_b$)
        $N_{shm\_comb}$=$N_{shm\_a}$+$N_{shm\_b}$;
        $N_{reg\_comb}$=$N_{reg\_a}$+$N_{reg\_b}$;
35:     $N_{warp\_comb}$=$N_{warp\_a}$+$N_{warp\_b}$;
        $N_{tblk\_comb}$=$N_{tblk\_a}$+$N_{tblk\_b}$;
        $N_{tblk\_SM\_comb}$=$N_{tblk\_SM\_a}$+$N_{tblk\_SM\_b}$;
        $U_{gpu\_b\_comb(a,b)} = \frac{B_{max\_mem} \cdot \sum_{i=a,b} N_{inst\_i}}{IPC_{max\_r} \cdot N_{SM} \cdot \sum_{i=a,b} R_{mem\_i} \cdot N_{inst\_i} \cdot N_{mem\_i}}$        ▷ Derived from Equation (7)
        return $K_{comb}$        ▷ Virtual "kernel" with combined profile
40: end function
```

Figure 1: Pseudocode: Concurrent Kernel *Symbiotic* Algorithm

Figure 1 presents the pseudocode of the proposed *symbiotic* heuristic, which is based on greedy method and implemented with C. All symbols used in the algorithm are defined in Table 2. The algorithm sequentially selects kernels based on a pre-calculated *Symbiosis Score*. Function SymbiosisScoreGen(K_M, K_N, PR) computes the score between every kernel pair taken from the set K_M and K_N respectively. The resultant score matrix is two dimensional or one dimensional depending on the input dimensions. For every kernel pair, the resulting SM hardware resources that remain available add to the score, shown in lines 19-22 in Figure 1. For all kernels to be scheduled, kernel pairs resulting in a lower and balanced usage of all four resource metrics are calculated with the highest *Symbiosis Score*. This allows more subsequent kernels to co-execute within the same *execution round*. On the other hand, a score component is also given by comparing the combined GPU resource utilization balance $U_{gpu\_b\_comb(i,j)}$ with the optimum (desired) balance U_{gpu_b}. $U_{gpu\_b\_comb(i,j)}$ is derived in function ProfileCombine(K_a, K_b) according to Equation (7) from the previous modeling analysis. The score decision is only considered when both kernels are of the opposing resource (compute/memory) characteristics. (For example, K_i is compute-bound while K_j is memory-bound.) Here we set a customized deviation limit $U_{gpu\_b\_devlim}$ to denote the maximum number of times that $U_{gpu\_b\_comb(i,j)}$ is greater or smaller than U_{gpu_b}. In other words, a score of zero is given if $U_{gpu\_b\_devlim}$ is exceeded on either side. Otherwise, an extra *Symbiosis Score* is given as shown in lines 23-29 based on the closeness of $U_{gpu\_b\_comb(i,j)}$ to the optimum balance requirement U_{gpu_b}.

The main algorithm takes the set of concurrent kernels to be reordered along with required profiling results as inputs. (Profiling results are collected through the CUDA profiler). Initially, the *SymbiosisScoreMatrix* is built to analyze the pair-wise resource-based *Symbiosis Score* among all kernels. The algorithm continues for each *execution round r*, the pair of kernel with the highest *Symbiosis Score* is selected for *symbiosis* within current *execution round r*, denoted by the set Rd_r. Note that for each *symbiotic* kernel pair, we order the pair decreasingly by the shared memory usage since this allows the kernel with more N_{shm_i} to release the resource sooner. In order to evaluate the *symbiosis* of current kernel pair with other unscheduled kernels if resource allows, the kernel pair is virtually combined by profile into a single virtual kernel K_{comb} with the function ProfileCombine(). The virtual kernel K_{comb} tracks multiple resource utilizations of current *execution round*, which is taken into consideration when choosing the next kernel for *symbiosis*. Therefore, a single dimension matrix SymbiosisScoreVec[ ] in line 10 is created to demonstrate the *Symbiosis Score* between current *execution round* and all unscheduled kernels. Kernels continue to be incorporated into the round r as long as the SM hardware resource permits until a new *execution round r+1* needs to be opened. The algorithm repeats until all kernels are scheduled into Rd_r and outputs the kernel launch schedule from Rd_1 to Rd_r that are *symbiotically* optimized.

5. EXPERIMENTAL RESULTS

In this section, we use a series of application benchmarks with varied profiles to demonstrate the improvements of execution time and energy consumption through the proposed *symbiotic* algorithm. All experiments are conducted under a single GPU computing node of our Cray XK7 supercomputer. The GPU node of Cray XK7 consists of an NVIDIA Tesla K20 GPU, a single socket AMD Opteron 6272 with 16 cores @2.1GHz and 16GB system memory. The NVIDIA K20 GPU is composed of 13 SMs @705 MHz with 4.8GB global memory @2600 MHz and allows up to 32 concurrent kernels to be executed. All benchmark results are collected under Cray Linux environment with CUDA 5.0. All resource metrics are experimentally collected by CUDA profiler for each kernel. Note that all the constant parameters from upper side of Table 2 are accordingly derived for K20 as the following [17]: N_{SM}=13; N_{reg_SM}=64K; S_{shm_SM}=48K; N_{warps_SM}=64; N_{tblk_SM}=16; IPC_{sp}=6; IPC_{dp}=2; IPC_{mem}=1;

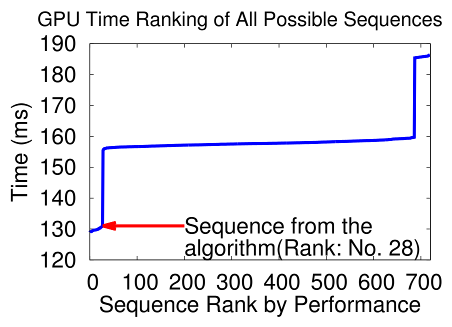

Figure 2: Performance: *EpShm-6*

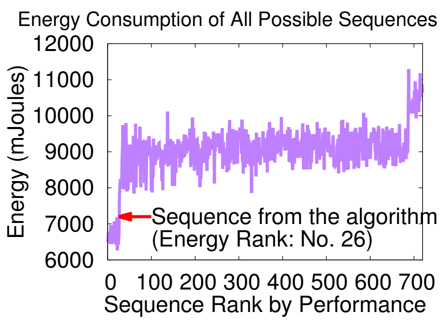

Figure 3: Energy: *EpShm-6*

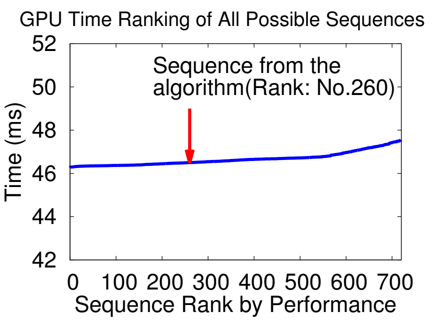

Figure 4: Performance: *EpGrid-6*

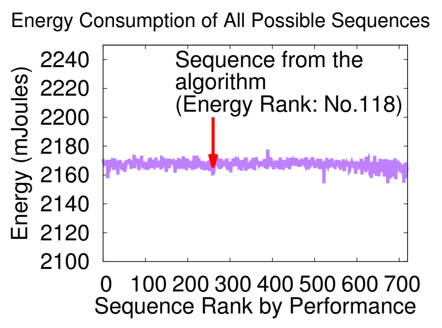

Figure 5: Energy: *EpGrid-6*

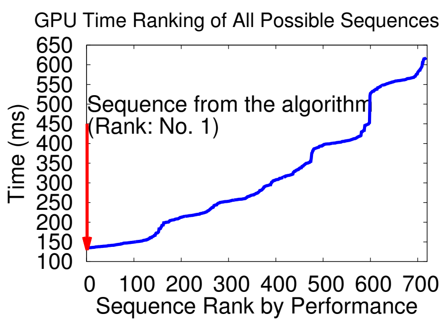

Figure 6: Performance: *BsBlk-6*

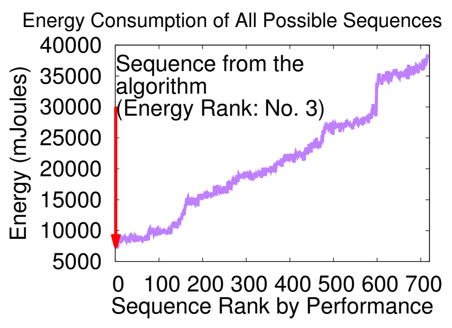

Figure 7: Energy: *BsBlk-6*

Table 3: Benchmark Profiles

Experiment	**Constant Metrics**	**Variables Across Kernels**
EpShm-6	$U_{gpu\_b\_i}$=0.224, N_{tblk_i}=26, $N_{thd\_per\_tblk\_i}$=128	N_{shm_i} = 12K, 16K, 20K, 22K, 26K, 48K
EpGrid-6	$U_{gpu\_b\_i}$=0.224, $N_{shm\_i} = 0$, $N_{thd\_per\_tblk\_i}$=128	$N_{warp\_SM\_i}$ = 8, 16, 24, 32, 40, 48 (N_{tblk_i} = 26, 52, 78, 204, 130, 156)
BsBlk-6	$U_{gpu\_b\_i}$=1.361, $N_{shm\_i} = 0$, N_{tblk_i}=26	$N_{warp\_SM\_i}$ = 4, 8, 12, 16, 32, 64 ($N_{thd\_per\_tblk\_i}$ = 64, 128, 256, 512, 768, 1024)
EpBsWarp-6	$N_{shm\_i} = 0$	3 EP kernels with $N_{warp\_SM\_i}$=8, $U_{gpu\_b\_i}$=0.224, 3 BS kernels with $N_{warp\_SM\_i}$=24, $U_{gpu\_b\_i}$=1.361
EpBsShm-6	—	3 EP with N_{shm_i}=16K,24K,48K 3 BS with N_{shm_i}=16K,24K,48K
EpBsEsSw-8	—	EP, BS, ES and SW kernels, 2 each

B_{max_mem}=302 Bytes/cycle. In our experiments, we use U_{gpu_b}=1 as the targeting optimum GPU utilization balance.

For power measurement, we utilize the on-board power measurement feature of the K20 with NVML[16] support. NVML provides the programmers with the API to check current power draw from the GPU sensor. Here we use a single POSIX thread (Pthread) that keeps polling the current power draw with a preset frequency (100Hz in our experiments) during the period of kernel execution. The Pthread also accumulates the total energy consumption over the time of concurrent kernel execution. We will mainly use the metric of total energy consumption for further analysis. On the performance side, we are mainly focused on the total execution time of a given concurrent kernel launch order.

5.1 Effectiveness of the Symbiotic Algorithm on Performance and Energy Optimization

To demonstrate the achievable reduction on both execution time and energy consumption of a kernel order with our *symbiotic* algorithm, all of our following experiments evaluate the concurrent execution and corresponding energy consumption of all possible kernel orderings. In other words, we conduct our experiment within the permutation (solution) space of kernel launch ordering and find the ranking of the kernel order given by our algorithm in terms of performance and energy consumption respectively. Initially, to demonstrate the effectiveness of the proposed algorithm on different resource metrics, we conduct five experiments, each of which consists of six concurrent kernels. We use NAS parallel Benchmarks (NPB) kernel EP [12] (M=24, $U_{gpu\_b\_i}$<1) and the European option pricing benchmark BlackScholes (BS) [3] (4M options, $U_{gpu\_b\_i}$>1) to represent memory-bound and compute-bound respectively. The benchmark profile of each experiment is listed in Table 3. Figure 2 to Figure 13 show the performance (execution time) and energy consumption (mJoules) for the permutation space of all six benchmarks. The performance and energy consumption of all possible sequences for each experiment are ranked with the execution time for each of the figures. As shown from Figure 2 to Figure 13, we are able to observe that the total energy consumption in general increases proportionally with the execution time. Therefore, the kernel *symbiosis* provided by our algorithm optimizes performance and energy simultaneously. Detailed results of each benchmark are explain as the following.

- *Single Application (constant $U_{gpu\_b\_i}$), varying shared memory usage only*: The experiment *EpShm-6* consists of six EP kernels that varies only the shared memory usage. We use *EpShm-6* to demonstrate the algorithm's effectiveness on shared memory resource conflicts among kernels. Figure 2 plots the performance

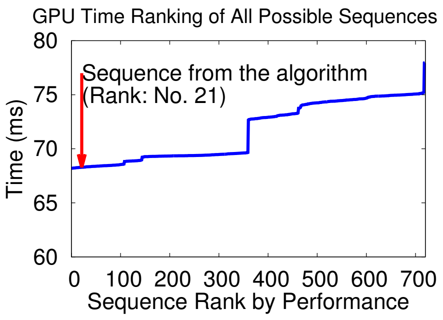

Figure 8: Performance: *EpBsWarp-6*

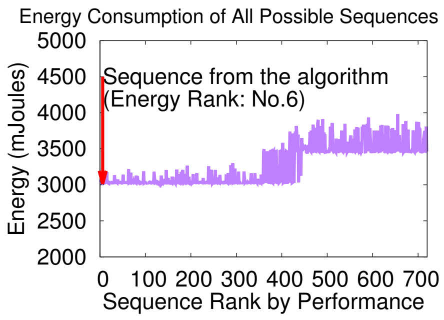

Figure 9: Energy: *EpBsWarp-6*

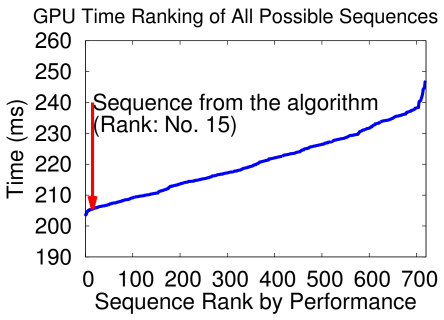

Figure 10: Performance: *EpBsShm-6*

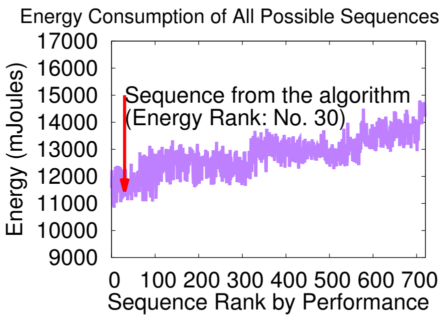

Figure 11: Energy: *EpBsShm-6*

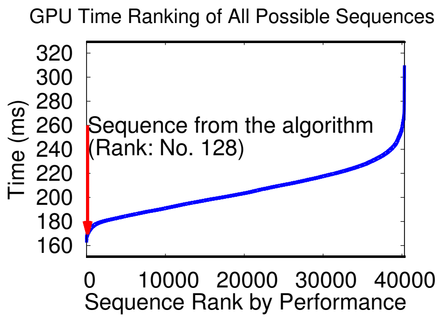

Figure 12: Performance:*EpBsEsSw-8*

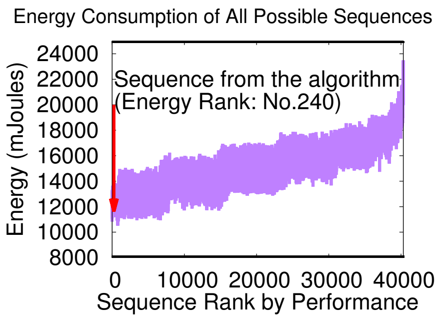

Figure 13: Energy: *EpBsEsSw-8*

of *EpShm-6*'s permutation space ranked by the total execution time while Figure 3 demonstrated the measured total energy consumption of each corresponding sequence of Figure 2. Both figures demonstrate that our algorithm can produce the concurrent kernel launch schedule with 96.1% and 96.4% percentile rank for performance and energy respectively.

- *Single Application (Constant $U_{gpu\_b\_i}$), varying warp usage only*: The experiment *EpGrid-6* varies the warp usage by only changing the kernel grid size among kernels whereas *BsBlk-6* instead varies only the block size alone. Our purpose of using both *EpGrid-6* and *BsBlk-6* is to demonstrate the impact of kernel launch order on both performance and energy consumption when kernel warp usage is varied by grid and block size respectively.
  - *Through varying grid sizes*: As shown in Figure 4 and 5, for *EpGrid-6*, both performance and energy stay approximately the same for all possible launch orders. This is because all six kernels have identical thread blocks. As discussed earlier in Section 3, the composition of thread blocks for each *execution round* as well as the number of *rounds* are the same regardless of the launch order. Thus, results and ranking shown in Figure 4 and 5 are merely to demonstrate this special scenario that kernel launch order does not matter.
  - *Through varying block sizes*: However, under varied warp usage of *BsBlk-6* due to different block sizes, both performance and energy are greatly impacted by the launch order, as shown in Figure 6 and 7. In this case, our algorithm provides the kernel order with 100% percentile rank in performance and 99.6% in energy.
- *Two Applications, varying warp usage only across the two applications*: The next experiment, *EpBsWarp-6* tests two different kernels with different warp usage and compute/memory utilization balance ($U_{gpu\_b\_i}$). Figure 8 and 9 demonstrate that our algorithm provides the performance and energy percentile rank of 97.1% and 99.2% for *EpBsWarp-6*.
- *Two Applications, varying warp usage across the two applications; varying shared memory usage across the 3 kernels of each application*: In the *EpBsShm-6* experiment, the effect of varying the shared memory is additionally further factored in compared with *EpBsWarp-6*. As shown in Figure 10 and 11, the sequence delivered by the algorithm achieves 97.9% and 95.8% rank in performance and energy respectively.
- *Four Applications, varying all metrics across 8 kernels*: We further conduct a more generic benchmark with four applications from different fields: the Electrostatics (ES) algorithm (40K atoms) from Visual Molecular Dynamics [2], the Smith Waterman algorithm (SW) plus BS and EP. The experiment of *EpBsEsSw-8* is composed of 2 kernels of each application with a total of 8 kernels. Each kernel of the eight is varied with each other for all metrics including N_{reg_i}, N_{shm_i}, N_{warp_i} (thus $N_{warp\_SM\_i}$), $N_{tblk\_SM\_i}$ and $U_{gpu\_b\_i}$. Both Figure 12 and 13 demonstrate near-optimal performance and energy results for *EpBsEsSw-8* due to the kernel *symbiosis* provided by our algorithm. In other words, within the 40,320 possible kernel launch orders, the proposed algorithm provides the sequence with 99.7% performance rank and 99.4% energy rank. Additionally, Figure 14 and 15 depict the distribution

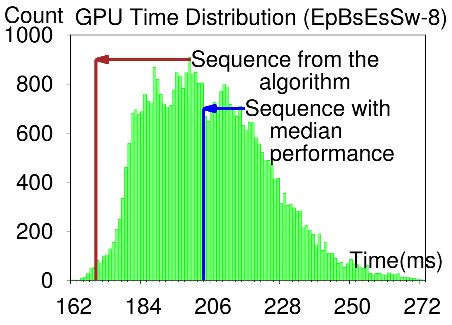

Figure 14: Time Distribution: *EpBsEsSw-8*

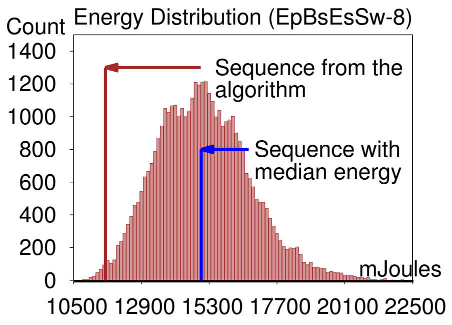

Figure 15: Energy Distribution: *EpBsEsSw-8*

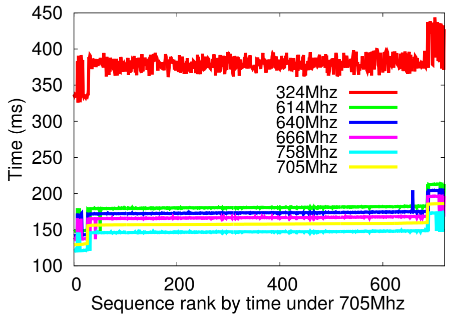

Figure 16: DVFS & Time: *EpShm-6*

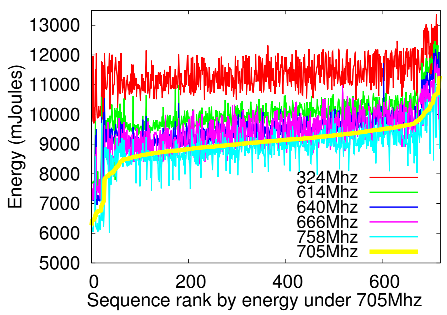

Figure 17: DVFS & Energy:*EpShm-6*

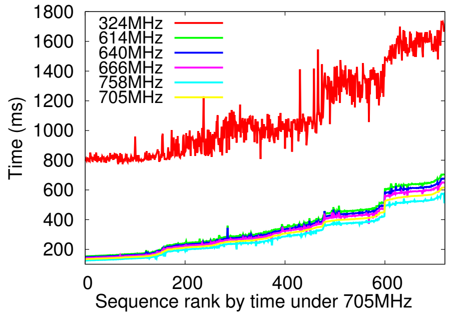

Figure 18: DVFS & Time: *BsBlk-6*

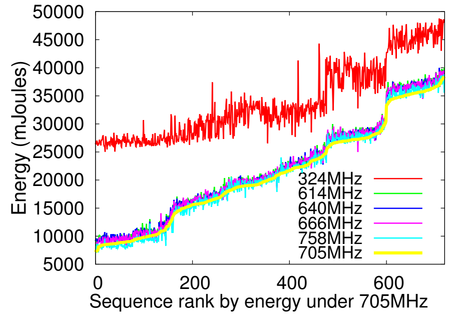

Figure 19: DVFS & Energy: *BsBlk-6*

Table 4: Comparisons of Benchmark Results

Experiment	**Performance Gain**			**Energy Improvement**		
	Dev. from optimal	Over median	Over worst	Dev. from optimal	Over median	Over worst
EpShm-6	1.64%	1.204	1.424	12.88%	1.263	1.598
BsBlk-6	0.00%	2.047	4.627	0.26%	2.738	5.231
EpBsWarp-6	0.15%	1.064	1.142	0.11%	1.023	1.320
EpBsShm-6	1.09%	1.070	1.202	5.54%	1.118	1.293
EpBsEsSw-8	4.43%	1.201	1.826	9.61%	1.292	2.021

of GPU time and energy consumption respectively. By comparing the algorithm sequence and the median sequence, we demonstrate that our algorithm has 50% of the probability to provide 20.1% performance gain and 29.2% reduction on energy consumption.

Table 4 further summarizes the performance and energy result comparisons for all the five benchmarks that are within the scope of our algorithm. In Table 4, we compare the algorithm results with the optimal (best ranking), median ranking and worst ranking sequence for both performance and energy. The deviation from the optimal, the performance/energy improvement over the median and worst results are presented. As we can see, all experiments demonstrate the near-optimal scheduling results of the proposed algorithm on reducing both execution time and total energy consumption for concurrent kernels, with up to 4.63 speedups and 5.23 times improvement on energy saving.

5.2 The Impact of DVFS on Performance and Energy for Concurrent Kernels

We further conduct a series of benchmarks to experimentally study the impact of frequency scaling on both performance and energy consumption. It is worth noting that our study is primarily focused on the impact of DVFS over concurrent many kernels instead of the single kernel scenario discussed in [5]. As Table 1 describes the supported GPU/memory frequencies that can be set through NVML [16], we carry out the similar performance and energy evaluations for two previously mentioned benchmarks: *EpShm-6* (memory-bound) and *BsBlk-6* (compute-bound). Both benchmarks are evaluated through the kernel order solution space for performance and energy consumption under all supported frequencies. Note that the GPU clock needs to be set according to the two supported memory clocks as shown in Table 1. Figure 16 and 17 respectively plot the execution time and energy consumption of *EpShm-6* under all possible GPU frequencies, while Figure 18 and 19 show the similar plots for *BsBlk-6*. From Figure 16 to 19, we observe that the highest GPU frequency produces the best performance and lowest energy consumption in general for both compute-bound and memory-bound workloads composed of concurrent kernels. While lowering the GPU frequency can help reducing the power, it instead increases the total energy consumption for concurrent kernels. Note that due to the limited GPU/memory clock pairs of K20, shown in Table 1, lowering the GPU clock to 324MHz forces the memory clock to be lowered to 324MHz as well. In other words, currently it is not possible to separately lower the GPU or memory frequency alone to 324MHz without affecting the other. This is why we are not able to provide the DVFS setting of high GPU clock / low memory clock for compute-bound benchmark (BsBlk-6) and low GPU clock (324MHz) / high memory clock for memory-bound benchmark (EpShm-6) as well. Therefore, due to the potential DVFS limitation, over-clocking on the K20 in general provides the best energy efficiency for concurrent kernel execution. For instance, over-clocking from 705MHz (stock) to 758MHz achieves 7.06% and 7.18% performance gain with

2.7% and 2.1% energy saving for the algorithm-derived kernel sequence of *EpShm-6* and *BsBlk-6* respectively.

As a summary, to optimize for both performance and energy consumption of concurrent kernels, we demonstrate that the proposed *symbiotic* algorithm can provide the near-optimal kernel launch schedule with significant performance gain and energy saving compared with a naive schedule. With the focus on the execution multiple concurrent kernels, our experimental results also demonstrate that over-clocking through DVFS further improves both performance and energy consumption while under-clocking surprisingly does the opposite due to the limited clock settings through DVFS. While our DVFS experimental study can help the programmers make optimization decisions under specific performance and energy constraints, the proposed *symbiosis* approach can be effortlessly adopted by GPU programmers with minimal code change to achieve optimized performance and energy consumption for concurrent kernels as well as make over-clocking unnecessary.

6. CONCLUSION

In this paper, we presented an algorithm-based performance and energy optimization technique, which is geared towards the computing phase of concurrent kernels in the GPU-based computing, We focused our analysis on reducing the GPU resource conflicts among kernels and developed necessary resource metrics with modeling analysis. Based on the analysis, a concurrent kernel *symbiotic* algorithm is proposed to derive a performance/energy-aware kernel launch schedule. The proposed algorithm is able to achieve an improved overall GPU SM resource utilization among all concurrent kernels with a well-balanced utilization on both the compute instruction throughput and memory bandwidth. The experimental results and analysis on the latest NVIDIA K20 GPU demonstrated that our algorithm is able to provide the kernel launch schedule with significant performance and energy benefits with near-optimal results. Further experimental analysis by applying DVFS on concurrent kernels provides the programmers with optimization insights on its possible impacts on performance and energy consumption. To the best of our knowledge, the work we have presented in this paper is the first to provide a kernel reordering and *symbiosis* solution for concurrent GPU kernels and our algorithm-based approach can be readily introduced into existing GPU programming environments with minimal risk.

7. ACKNOWLEDGMENT

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. IIP-1161014 and IIP-1230815.

8. REFERENCES

[1] Top 500 Supercomputer Sites Webpage, Jan. 2014. `http://www.top500.org`.

[2] Visual Molecular Dynamics Program Webpage, Jan. 2014. `http://www.ks.uiuc.edu/Research/vmd/`.

[3] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, 81(3):637–654, 1973.

[4] S. Dreßler and T. Steinke. Energy consumption of CUDA kernels with varying thread topology. *Computer Science-Research and Development*, 27(3):1–9, Aug. 2012.

[5] R. Ge, R. Vogt, J. Majumder, A. Alam, Burtsch.Martin, and Z. Zong. Effects of dynamic voltage and frequency scaling on a K20 GPU. In *2nd International Workshop on Power-aware Algorithms, Systems, and Architectures*, 2013.

[6] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *Proc. 4th USENIX conference on Hot Topics in Parallelism*. USENIX Assoc., 2012.

[7] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *Computer Architecture Letters*, 8(1):25–28, 2009.

[8] S. Hong and H. Kim. An integrated gpu power and performance model. In *Proc. 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM.

[9] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and performance characterization of computational kernels on the GPU. In *Proc. Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 221–228. IEEE, 2010.

[10] Khronos OpenCL Working Group. *The OpenCL Spec V2.0*, July 2013.

[11] F. Lu, J. Song, F. Yin, and X. Zhu. GPU computing using concurrent kernels: A case study. In *Proc. 2nd World Congress on Computer Science and Information Engineering (CSIE 2011)*, pages 173–181, 2011.

[12] M. Malik, T. Li, U. Sharif, R. Shahid, T. El-Ghazawi, and G. Newby. Productivity of GPUs under different programming paradigms. *Concurrency and Computation: Practice and Experience*, 24(2):179–191, 2012.

[13] P. Micikevicius. GPU performance analysis and optimization, 2012. Available online on `http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf`.

[14] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka. Statistical power modeling of GPU kernels using performance counters. In *Proc. Green Computing Conference, 2010 International*, pages 115–122. IEEE, 2010.

[15] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012. Ver. 1.0, Available online on `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[16] NVIDIA. *NVML API Reference Manual*, 2012. Ver. 4.304.55, Available online on `https://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDADownloads/NVML_cuda5/nvml.4.304.55.pdf`.

[17] NVIDIA. *NVIDIA CUDA C-Programming Guide V5.5*, July 2013.

[18] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Proc. 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 407–418, 2013.

[19] A. Snavely, N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Explorations in symbiosis on two multithreaded architectures. In *Workshop on Multithreaded Execution And Compilation (MTEAC)*, Jan. 1999.

[20] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 234–244, New York, NY, USA, 2000. ACM.

[21] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *Proc. Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & 2010 IEEE/ACM Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 344–350. IEEE, 2010.